DURAL : an Extended Prolog Language (Mathematical Methods in Software Science and Engineering : Second Conference)

AUTHOR(S):

GOTO, SHIGEKI

数理解析研究所講究録
396巻 1980年 170-189

DURAL: an extended Prolog language

Shigeki Goto
Musashino Electrical Communication Laboratory
Nippon Telegraph and Telephone Public Corporation
3-9-11, Midori-cho, Musashino-shi
Tokyo 180 JAPAN

ABSTRACT

Prolog is a programming language based on the predicate calculus. Each Prolog statement takes the form of a special logical formula, called a Horn clause, which can be interpreted operationally as a procedure declaration.
This paper proposes a new programming language DURAL as an extension of the Prolog language. DURAL takes advantage of the relative Horn clause to speed up execution of programs.
The main features of DURAL are the following:
1) The relative Horn clause represents the clause containing executable predicates.
2) Program synthesis capability can be easily implemented by means of the relative Horn clause.
3) Modal symbols are introduced to classify the clauses.

KEY WORDS & PHRASES

DURAL, extensional database, intensional database, modal logic, predicate calculus, program synthesis, programming language, Prolog, query language, relative Horn clause, resolution principle.

TABLE of CONTENTS

1. Prolog --- Horn clause

Prolog was first proposed by Kowalski[1], and implemented at Université d'Aix-Marseille (Battani & Meloni[2]). This section introduces the original Prolog language and shows the flexibility of Prolog by several examples. Example 1 is a simple program written in Prolog. This simple example is chosen for convenience of explanation. In fact, many working programs have been written in Prolog. They include natural language understanding systems, formula manipulation and symbolic integration systems, and a STRIP-like problem solver.

EXAMPLE 1. Addition in Prolog

```
1   +(ADD 0 *Y *Y)
2   +(ADD (S *X) *Y (S *Z)) -(ADD *X *Y *Z)
3   -(ADD 2 3 *Z)
```

Before inspecting example 1, note that Prolog's syntax is given in modified BNF notation. In this paper, the Prolog interpreter is written in Lisp and the Lisp "atom" is used as the primitive constituent of Prolog syntax.
In definition 1, a pair of square brackets [,] enclose the optional items. An ellipsis "..." indicates a list consisting of the preceding item, e.g., <A>... means <A> | <A><A> | <A><A><A> and so on. A vertical bar "|" means "or", and separates alternative items.

DEF. 1 Syntax of Prolog

<program> ::= <statement>...

<statement> ::= <Horn clause>

<Horn clause>::=[<positive literal>][<negative literal>...]

<positive literal> ::= +<atomic formula>

<negative literal> ::= -<atomic formula>

<atomic formula> ::= (<predicate> <term>...)

<predicate> ::= <Lisp atom>

<term> ::= <Lisp atom> | <variable> | (<function><term>...)

<variable> ::= *<Lisp atom>

<function> ::= <Lisp atom>

<Lisp atom> is the literal atom in the programming language Lisp, and is not defined here.

According to the definition above, a Horn clause has at most one positive literal and a finite number, possibly zero, of negative literals. Table 1 divides Horn clauses (Prolog statements) into four cases. In each case, the Horn clause

can be interpreted operationally as a procedure declaration.

Table 1. Horn clause

\neg / pos	0	not 0
0	□	-B1-B2...-Bn
1	+A	+A-B1-B2...-Bn

Procedural interpretation:

□ STOP statement

i.e. procedure without a name or body

□ stands for an empty clause.

+A-B1-B2... PROCEDURE declaration

+A: procedure name and variable list

-B1-B2...: procedure body which calls B1, B2, ...

-B1-B2... EXECUTE B1, B2, ... (procedure without a name)

called a goal statement in Prolog

+A PROCEDURE without a body

This type of statement is not meaningless, since the variable list may contain various terms.

In example 1, three types of statement appear. A normal program has exactly one goal statement. The goal statement changes its shape during the program execution, and finally takes the form of a STOP statement when the program terminates.

EXAMPLE 1. (listed again)

```
1   +(ADD 0 *Y *Y)
2   +(ADD (S *X) *Y (S *Z)) -(ADD *X *Y *Z)
3   -(ADD 2 3 *Z)
```

ADD is a predicate and (ADD a b c) means a+b=c; S is the successor function, namely (S x)=x+1. The program execution is performed by Algorithm 1 below.

ALGORITHM 1. (Prolog)

L means "line". The execution starts at line 1.

L	Action or Test	Next Line: Succeed	Next Line: Fail
1	Find a goal statement "G"	2	abort
2	G is □ (STOP)	stop	3
3	G:={Resolve G against another clause}	2	4
4	G:={The previous G}, backtracking	2	abort

The abortion in line 1 means there is no goal statement. The abortion in line 4 means G cannot be resolved against any clause in the program.

The backtracking is done by the stack mechanism.

The resolution in line 3 represents the input resolution. The adoption of the input resolution is supported by the following proposition.

PROP. 1 If S is an unsatisfiable Horn set, then there is an input refutation of S.

The following example shows the execution of the

above-mentioned program.

EXAMPLE 2. Execution of example 1

A number is automatically converted into the form (S (S ...0)), e.g. 2 -> (S (S 0)).

```
L1:  G:= -(ADD (S (S 0)) (S (S (S 0))) *Z)
L2:  G is not □.
L3:  G:= -(ADD (S 0) (S (S (S 0))) *Z)
L2:  G is not □.
L3:  G:= -(ADD 0 (S (S (S 0))) *Z)
L2:  G is not □.
L3:  G:= □
L2:  G is □.        normal termination
```

Although the execution is terminated normally, the answer 5 (=2+3) is lost, since there is no output statement. To implement the output statement, a built-in predicate "OUT" is introduced. It is assumed that the following statements (clauses) are built into the Prolog system.

```
+(OUT *X1)
+(OUT *X1 *X2)
     .
     .
+(OUT *X1 *X2 ... *Xn)
     .
     .
```

If a goal statement contains a negative literal -(OUT T1 T2...Tm), where Ti is any term, the resolution is always successful due to the built-in clause +(OUT *X1 *X2...*Xm). The special OUT resolution has the side effect that T1, T2, ..., Tm are printed on the terminal. Example 3 illustrates the OUT predicate.

EXAMPLE 3. Subtraction using a built-in predicate OUT

```
  1   +(ADD 0 *Y *Y)
  2   +(ADD (S *X) *Y (S *Z)) -(ADD *X *Y *Z)
  3   -(ADD *X 3 5) -(OUT *X)                    <=== goal
```

A number is converted, e.g. 3 -> (S (S (S 0))).

A comma "," indicates line continuation.

```
L1:   G:= -(ADD *X (S (S (S 0))) (S (S (S (S (S 0))))))  ,
          -(OUT *X)
L2:   G is not □.
L3:   G:= -(ADD *X (S (S (S 0))) (S (S (S (S (S 0))))))  ,
          -(OUT (S *X))
L2:   G is not □.
L3:   G:= -(ADD *X (S (S (S 0))) (S (S (S (S 0)))))  ,
          -(OUT (S (S *X)))
L2:   G is not □.
L3:   G:= -(OUT (S (S 0)))
L2:   G is not □.
L3:   G:= □          side effect: (S (S 0)) is printed.
L2:   G is □.        normal termination
```

The output (S (S 0)) is also converted to 2.

Above example demonstrates that subtraction can be performed through the same clauses 1 and 2 as for addition in example 1. In fact, many different goal statements can be executed using clauses 1 and 2 (see below).

EXAMPLE 4. A variety of goal statements

```
-(ADD 2 3 *Z) -(OUT *Z)                    addition:    *Z=5
```

-(ADD 2 *Y 5) -(OUT *Y)	subtraction: *Y=3
-(ADD *X 3 5) -(OUT *X)	subtraction: *X=2
-(ADD 2 3 5)	normal termination
-(ADD 2 3 6)	abnormal termination
-(ADD 2 *Y *Z) -(OUT *Y *Z)	*Y=*Y, *Z=(S (S *Y))
-(ADD *X 3 *Z) -(OUT *X *Z)	*X=0, *Z=3
-(ADD *X *Y 5) -(OUT *X *Y)	*X=0, *Z=5
-(ADD *X *Y *Z)-(OUT *X *Y *Z)	*X=0, *Y=*Y, *Z=*Y

The goal statement -(ADD 2 3 6) causes an abnormal termination, since 2+3 is not 6. -(ADD 2 *Y *Z) and -(ADD *X *Y *Z) have symbolic (not numerical) answers. It should be noted that even if the goal statement has more than one answer, only the one that is found first is printed. It is because of the character of the resolution principle, and will be discussed further in section 3.

As a final example in this section, example 5 shows a shuffled program that is a mixture of the addition program and the classical syllogism "Socrates is mortal". The program works well and the answer is *W=Socrates. It can also behave as an addition program if the goal statement takes the form of -(ADD 2 3 *Z).

EXAMPLE 5.

```
1   +(MAN Socrates)
2   +(ADD (S *X) *Y (S *Z)) -(ADD *X *Y *Z)
3   +(MORTAL *X) -(MAN *X)
```

```
4    +(ADD 0 *Y *Y)
5    -(MORTAL *W) -(OUT *W)
```

As was shown in examples 4 and 5, Prolog has a powerful flexibility which makes it suitable for the artificial intelligence applications. On the other hand, the execution speed is rather slow because of the repeated search for resolvable clauses. The next section treats techniques for speeding up execution.

2. DURAL --- relative Horn clause

2.1 Fast version clause

Methods for improving the efficiency of Prolog programs are divided into two categories.

i) External augmentation: Georgeff[3] has proposed supplying control information in the form of a regular expression. In example 1, a regular expression (2)*1 directs the addition to start at clause 2 and use it several times, and finally to execute clause 1 once. Similar techniques are discussed elsewhere in the literature and are omitted from this paper.

ii) Internal reinforcement: A fast version of the addition program can be written as follows, where "PLUS" is a Lisp function.

```
+(ADD *X *Y (PLUS *X *Y))
-(ADD 2 3 *Z) -(OUT *Z)        <=== goal
```

The goal statement is executed and terminated instantly. It prints the answer *Z=(PLUS 2 3). If the answer is evaluated in Lisp, *Z is equal to 5. This paper shows that the fast version technique is powerful enough to write useful programs in Prolog.

It is worth noting here that the fast version clause is less flexible than the ordinary one. Example 6 below shows that the fast clause +(ADD *X *Y (PLUS *X *Y)) cannot be used for subtraction, since (PLUS *X *Y) is a term and cannot be unified with a constant "5".

EXAMPLE 6.

```
+(ADD *X *Y (PLUS *X *Y))
-(ADD 2 *Y 5) -(OUT *Y)     <=== goal, abnormal termination

+(ADD *X *Y (PLUS *X *Y))
-(ADD *X 3 5) -(OUT *X)     <=== goal, abnormal termination
```

2.2 Executable predicate

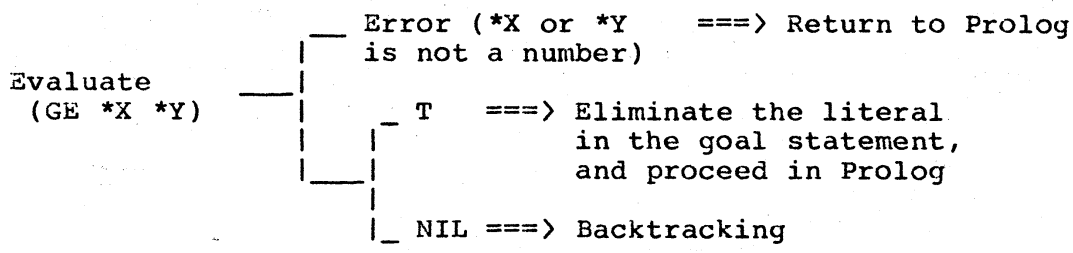
Lisp predicates, as well as Lisp functions, can be used in the fast version programs. In example 7, it is possible to replace clauses 3 and 4 with a predicate GE, if it is declared that GE is executable and the corresponding Lisp routines 3′ and 3′′ are prepared.

EXAMPLE 7.

```
1   +(RESULT *X 1) -(GE *X 10)
2   +(RESULT *X 0) -(GE 10 *X)
3   +(GE (S *X) (S *Y)) -(GE *X *Y)     <--- to be replaced
```

```
  4    +(GE *X 0)                          <--- to be replaced
  3'   (DE GE (X Y)                       <--- Lisp routine
           (OR (EQ X Y) (GREATERP X Y)) )
  3''  (DE S (X) (ADD1 X))                 <--- Lisp routine
```

If the declaration is issued, the negative literal -(GE *X *Y) does not search for the corresponding positive literal any more, but evokes the evaluation in Lisp.



The return to Prolog is carried out through the ERRSET facility in Lisp. When the value of (GE *X *Y) is T, the execution proceeds successfully. If the value is NIL, the value of -(GE *X *Y) becomes T. This means the goal statement cannot be proved by resolution, and it brings the execution into a backtracking mode as in line 4 in algorithm 1.

In fact, the built-in predicate in the previous section played a similar role. It can be regarded as an executable predicate and its corresponding Lisp routine causes the side effect.

If the executable predicate is to be used freely, the Prolog system must check whether the predicate is built-in or not. The present system DURAL saves checking by modifying the Prolog syntax. In the original Prolog syntax, a goal

statement contains only negative literals. In DURAL, a positive literal may appear in a goal statement, and it is treated as an executable literal. The syntax is extended to include positive literals in the goal statement, which is formally based on the relative Horn clause.

DEF 2. (Relative Horn clause) A set S of clauses defines a set of Horn clauses relative to M, where M is a setting, if and only if each clause of S has at most one literal false in M.

A setting is in effect a consistent set of literals from the Herbrand universe of S.

DEF 3. (Setting) Given a set S of clauses, a <u>setting</u> M for S is a (possibly empty) set of literals satisfying the following conditions:

i) every literal of M is an S-instance of a literal of S or an S-instance of the complement of a literal of S;

ii) if L is in M, then every S-instance of L is in M;

iii) M does not contain a complementary pair of literals.

An S-instance of a literal L is a literal L[t1/x1,...] where [t1/x1,...] denotes a substitution and the terms t1,... consist of the alphabet of S plus variable names. An S-generalization of a literal L is an S-instance L′ of some literal of S such that L is an S-instance of L′.

DEF 4. (Homogeneous setting; partition)

i) A setting M is homogeneous if and only if whenever L is

in M then every S-generalization of L is in M.

ii) A partition M is a setting M such that every S-instance of a literal of S is in M or has its complement in M.

A partition is by nature a homogeneous setting. The present DURAL system utilizes only a subset of the relative Horn clause.

DEF 5. (Extended DURAL syntax)

⟨statement⟩ ::= ⟨positive literal⟩[⟨literal⟩...] |
⟨goal statement⟩

⟨goal statement⟩ ::= □ | ⟨negative literal⟩[⟨literal⟩...]

⟨literal⟩ ::= ⟨positive literal⟩ | ⟨negative literal⟩

ALGORITHM 2

L means "line". The execution starts at line 1.

L	Action or Test	Next Line: Succeed	Next Line: Fail
1	Find a goal statement "G"	2	abort
2	G is □ (STOP)	stop	3
3	The left-most literal in G is positive	4	5
4	EXECUTE the literal in Lisp	2	5/6
5	G:={Resolve G against another clause}	2	6
6	G:={The previous G}, backtracking	2	abort

In line 4, three cases may occur:

Evaluate the positive literal +P

- Error in Lisp ===> (5) Return to DURAL resolution mode
- T ===> (6) The goal cannot be proved, since the goal itself will be T.
- NIL ===> (2) Success

The resolution in line 5 represents the input resolution.

PROP 2. (Loveland[4]) If S an unsatisfiable Horn set relative to a homogeneous partition M for S, then there exists an input refutation of S.

EXAMPLE 8. Executable predicate ANS

```
1   -(ADD 2 3 *Z) -(OUT *Z)     built-in OUT
2   -(ADD 2 3 *Z) +(ANS *Z)     executable ANS
```

The predicate ANS performs the same job as OUT. However, clause 2 is not logically equivalent to clause 1. In fact, clause 1 is transformed into the formula -[(ADD 2 3 *Z) & (OUT *Z)], which is the negation of the formula [(ADD 2 3 *Z) & (OUT *Z)] to be proved. This type of goal statement is used conventionally, since the resolution principle proves the formula by refutation. On the other hand, clause 2 is equivalent to (ADD 2 3 *Z) -> (ANS *Z), which reads "whenever *Z satisfies (ADD 2 3 *Z), it is the answer." The answering predicate ANS is often applied to question answering systems (Chan & Lee [5]). DURAL incorporates the answering predicate into the executable predicates.

2.3 Synthesis capability

Applying the program synthesis technique, a fast version program is constructed automatically from ordinary clauses. The theorem proving approach (Manna & Waldinger[6]) to program synthesis is effective, since each DURAL statement takes the form of a logical formula.

Moreover, as DURAL statements can be considered in intuitionistic logic, one can use the synthesis method based on Gödel's interpretation (Goto[7] , Sato[8]).

EXAMPLE 9.

```
1   +(ADD 0 *Y *Y)
2   +(ADD (S *X) *Y (S *Z)) -(ADD *X *Y *Z)
```

From 1 and 2, the Lisp function ADD is synthesized.

```
3   (DE ADD (*X *Y)
        (COND ((ZEROP *X) *Y)
              (T (S (ADD (SUB1 *X) *Y))) ))
```

Clauses 1 and 2 logically represent a mathematical induction which corresponds to recursive program 3. A more interesting example is shown below.

EXAMPLE 10.

```
1   +(REVERSE NIL NIL)
2   +(REVERSE (CONS *X *Y) (NCONC1 *Z *X)) -(REVERSE *Y *Z)
3   (DE REVERSE (*Y)
        (COND ((NULL *Y) NIL)
              (T (NCONC1 (REVERSE (CDR *Y)) (CAR *Y))) ))
```

Example 10 implies that a linear list can be treated analogously to a natural number.

natural number	linear list
0	NIL
(ZEROP *X)	(NULL *X)
(S *Y)	(CONS *X *Y)
(SUB1 *X)	(CDR *X)

In addition, the synthesis algorithm itself can be represented in DURAL. In example 11, GETC and PUTC are executable predicates. In the terminology of production systems, the clause in example 11 is a meta-rule, where the variable *P ranges over predicates.

EXAMPLE 11. Synthesis algorithm in DURAL

```
+(INDS *P) ,
    +(GETC (+(*P 0 *F *T0))) ,
    +(GETC (+(*P (S *E) *F *T2)-(*P *E *F *T1))) ,
    +(PUTC (+(*P *E *F (*P *E *F)))
           (*P (*E *F)
               (COND ((ZEROP *E) *T0)
                     (T subst[(*P (SUB1 *E) *F),*T1,*T2])))))
```

The goal statement -(INDS ADD) produces the function ADD in example 9.

3. DURAL --- modal symbol

3.1 Clause discrimination

There is no syntactical distinction between the fast version and the ordinary one. Sometimes it is necessary to distinguish them. For example, a goal statement -(ADD 2 *Y 5) +(ANS *Y) cannot be executed under the fast version clause +(ADD *X *Y (PLUS *X *Y)), but if the ordinary clauses +(ADD 0 *Y *Y) and +(ADD (S *X) *Y (S *Z)) -(ADD *X *Y *Z) coexist, the goal is achieved through the ordinary clauses. DURAL (Prolog) has an automatic backtrack facility, so one cannot confirm whether execution is

performed only in the fast version environment. To distinguish the clauses, the modal symbol is introduced.

DEF 6. (modal symbol in DURAL)

⟨negative literal⟩ ::= -⟨atomic formula⟩ |

- S ⟨atomic formula⟩

The symbol "S" is a sign of strongly provability. I.e., a literal - S (ADD 2 *Y 5) must be resolved against a fast clause exclusively, whereas -(ADD 2 *Y 5) may be resolved against any clause in the program. This settles the above problem.

A further application of the modal symbol is explained in the database field. The resemblance between relational database models and Prolog-like languages is well known (Fuchi[9], Futô, Darvas & Szeredi[10]). As is often the case, the extensional database (elementary facts) is allowed to change over time, whereas the intensional database (general laws) tends to remain fixed. DURAL facilitates the distinction between the time-varying part and the fixed part. The modal symbol "S" can be regarded as indicating fixed clauses.

Example 12 defines a path relation in a directed graph G. F1 and F2 are assumed to be fixed. The time varying part (1 to 5) represents arcs in G. A goal statement contains an executable predicate "QUERY" which is much like the predicate ANS and has two side effects:

i) It prints the answer;

ii) It pretends that the evaluation fails, and evokes a backtrack mechanism.

The usage of the backtrack mechanism for plural answers is discussed elsewhere (e.g., [10]). DURAL realizes the mechanism through the relative Horn clause.

EXAMPLE 12.

```
S1  +(PATH *X *X)
S2  +(PATH *X *Z) -(ARC *X *Y) -(PATH *Y *Z)
1   +(ARC 1 2)
2   +(ARC 1 4)
3   +(ARC 2 3)
4   +(ARC 4 2)
5   +(ARC 4 3)
6   -(PATH 4 *W) +(QUERY *W)        <=== goal
7   - S (PATH 2 *W) +(QUERY *W)     <=== goal
```

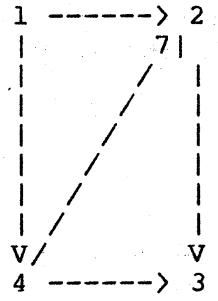


Fig.1 A directed graph G

The goal statement 6 will be answered by a set of nodes, {4, 2, 3}. Another goal, 7, must be carried out in the fixed part, and the answer is only {4} (using F1).

3.2 Alternative approach to executable predicates

In section 2.2, the executable predicate was introduced in the form of a positive literal. It is possible to represent the executability by prefixing a modal symbol "EX" to a literal.

-(ADD 2 3 *Z) - EX (ANS *Z)

Although there seems to be no essential difference between the two methods, the underlying logic is quite different. By the use of a modal symbol "EX", an interesting clause may be written.

+(ANS *Z) - EX (ANS *Z)

Since tautologies are not admitted as clauses in the resolution principle, any clause of the form +(P ...) -(P ...) must not appear in DURAL (Prolog). +(ANS *Z) - EX (ANS *Z) is not a tautology and is admitted as a clause. Note that if a tautology were admitted as a clause in DURAL (Prolog), it might cause an infinite loop.

Concluding Remarks

(1) DURAL has been written in INTERLISP, and run on DEC system 20. The size of the source program is about 660 lines when printed out neatly.

(2) Proposition 2 in section 2.2 can be strengthened by imposing a restriction on the resolution. The input resolution can be an ordered input resolution. The running DURAL (1) takes advantage of the ordering.

ACKNOWLEDGMENT

The author wishes to express his sincere thanks to Dr. K.Fuchi at Electrotechnical Laboratory for his guidance to the Prolog language.

REFERENCES

[1] R.Kowalski, Predicate Logic as Programming Language, IFIP-74, 569-574.

[2] G.Battani and H.Meloni, Interpréteur du langage de programmation PROLOG, Rapport de D.E.A. d'informatique appliquée, Groupe d'intelligence Artificielle Université d'Aix-Marseille.

[3] M.P.Georgeff, A Framework for Control in Production Systems, IJCAI-79, pp.328-334.

[4] D.W.Loveland, Automated Theorem Proving: A logical Basis, North-Holland, 1978.

[5] C-L.Chang and R.C-T.Lee, Symbolic Logic and Mechanical Theorem Proving, Academic Press, 1973.

[6] Z.Manna and R.J.Waldinger, Toward Automatic Program Synthesis, Comm. ACM, vol.14, no.3, pp.151-165, 1971.

[7] S.Goto, Program Synthesis from Natural Deduction Proofs, IJCAI-79, pp.339-341.

[8] M.Sato, Towards a Mathematical Theory of Program Synthesis, IJCAI-79, pp.757-762.

[9] K.Fuchi, private communication, 1977.

[10] I.Futó, F.Darvas and P.Szeredi, The Application of PROLOG to the Development of QA and DBM Systems, in LOGIC and DATABASES, pp.347-376, Plenum Press, 1978.